Generalisation of Recursive Doubling for AllReduce: Now with Simulation

Parallel Computing xxx (2017) xxx-xxx

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com

Generalisation of recursive doubling for AllReduce: Now with simulation

Martin Ruefenacht[?]*, Mark Bull, Stephen Booth

Edinburgh Parallel Computing Centre, University of Edinburgh, Edinburgh, Scotland, United Kingdom

ARTICLE INFO

Article history:
Received 15 January 2017
Received in revised form 27 June 2017
Accepted 17 August 2017
Available online xxx

Keywords:
AllReduce
MPI
Scalability
Collective
Recursive doubling
n-way
Message pipelining

ABSTRACT

The performance of AllReduce is crucial at scale. The recursive doubling with pairwise exchange algorithm theoretically achieves $O(\log_{?2} N)$ scaling for short messages with N peers, but is limited by improvements in network latency. A multi-way exchange can be implemented using message pipelining, which is easier to improve than latency. Using our method, recursive multiplying, we show reductions in execution time of between 8% and 40% of AllReduce on a Cray XC30 over recursive doubling. Using a custom simulator we further explore the dynamics of recursive multiplying.

1. Introduction

As supercomputers are pushed further towards exascale, the scaling behaviour of the algorithms used on them becomes ever more important due to the increased levels of parallelism in these systems. Both application layer and network layer algorithms are required to scale, in order to make optimal use of the hardware present. In addition, future interconnect hardware may provide more flexibility and compute capability compared to current hardware.

The AllReduce operation is one of the most important operations used on supercomputers [16]. Improving AllReduce will result in improved scalability for many important applications. Aside from performance the operation must provide consistent (i.e. bit-wise identical) results across all processes and executions, thereby ensuring correct and repeatable results. Applications which perform simulations using floating point arithmetic require consistent ordering of operations to achieve this.

The state of the art algorithm used for AllReduce operations, recursive doubling with pairwise exchange, scales as $O(\log_{?2} N)$. As ever larger computers are built this logarithmic behaviour will limit scaling behaviour and the only potential gain is by reducing network latency. Since network latency is a physically limited quantity and increasingly difficult to improve, another approach for performance gains must be found.

We seek an AllReduce algorithm which gives us consistency guarantees and performs better than the current recursive doubling with pairwise exchange algorithm for small messages. We show that using a model of message pipelining hardware, it is possible to extend the current recursive doubling with pairwise exchange algorithm (as used in MPICH [3]). Our method (recursive multiplying) reduces execution time, relative to that of recursive doubling, by between 8% and 40% on a Cray XC30.

* Corresponding author.
Email addresses: m.ruefenacht@ed.ac.uk, m.a.ruefenacht@gmail.com (M. Ruefenacht); m.bull@ed.ac.uk (M. Bull); s.booth@ed.ac.uk (S. Booth)

https://doi.org/10.1016/j.parco.2017.08.004
0167-8191/

The remainder of this work is organized as follows. In Section 2 the current state of the art algorithm for AllReduce is presented, including the specific details used in MPICH. Section 3 presents two models used to analyse the AllReduce operation in the context of message pipelining. The recursive multiplying algorithm is presented in Section 4. Experimental evidence for the theoretically better performance is demonstrated in Section 5. Section 7 presents related work for AllReduce. Finally, future work is discussed in Section 8.

2. Background

MPICH [3] is one of the primary MPI [2] implementations which is used as a base for many commercial implementations. Cray MPI is an example of an implementation using MPICH, which re-implements only the low level network interface, the Netmod interface [15]. This design allows ease of development when porting to different network architectures without having to rewrite higher level algorithms.

One such operation which is abstracted from the actual network is AllReduce. AllReduce is heavily used by many simulation applications and therefore high performance is important. In MPICH, recursive doubling with pairwise exchange is used for AllReduce when any of three conditions is met. The first is for small messages below 2KB. The second condition is when a user-defined operation is used to reduce the vector during each stage, so that non-associative operators are not erroneously optimized with other algorithms. The third condition is if the number of elements in the vector is below the nearest lower power of two of the process count. The implementation of AllReduce in MPICH was originally implemented by Thakur et al. [18,19].

Recursive doubling with pairwise exchange enables logarithmic scaling of $O(\log_{?2} N)$. Fig. 1 illustrates the communication pattern which is separated into multiple stages. During each stage, every process in the group sends to its corresponding peer appropriate for the current stage. This results in an AllReduce operation being able to be performed, because previous stages will have communicated data across subgroups. Between each communication round, a reduce operation is executed locally on each process to combine the received partial result and local result.

Each stage of the recursive doubling algorithm consists of pair-wise exchanges with combination of the results. These are implementations of the AllReduce operation over pairs of processors. The overall AllReduce operation is built up by recursively applying a series of smaller AllReduce operations over orthogonal sub-groups of processes, until the entire target group has been reduced. Pair-wise exchange and combination is the two processor version of a general algorithm for an AllReduce where each processor broadcasts its data to all other processors (the All-to-All communication pattern) and the results are reduced locally on every processor. Our recursive multiplying algorithm uses larger reductions (using this general algorithm) to reduce the number of communication stages required.

Recursive doubling requires a power of two group of processes, since each stage subdivides the group by two. With large numbers of processes, powers of two are sparse, so this is a strong limitation to the usability of the basic method. MPICH fixes the non-power-of-two problem by collapsing and expanding the group in two additional stages. Fig. 2 illustrates the solution for six processes. To determine the correct rank with which to communicate, a virtual to real transform has to be performed for every send operation to avoid a deadlock.

First, the next lowest power of two to the size of the collective is found by $p = 2^{\lfloor \log_2 N \rfloor}$, and the remainder is given by $r = N - p$. This establishes a usable size for the recursive doubling algorithm. All ranks i where i is even and $i < 2r$ send to their peer of rank $i + 1$. Then the recursive doubling algorithm with pairwise exchange is executed on the remaining active processes. The pseudocode is shown in Algorithm 1. Finally, the expansion stage is performed, in which the original processes which received data from their neighbour return the final result.

This fix allows the non-power-of-two case to be handled elegantly with minimal additional stages. While recursive doubling scales well, it is important to maximize the performance, since AllReduce is an important operation. Fundamentally the algo-

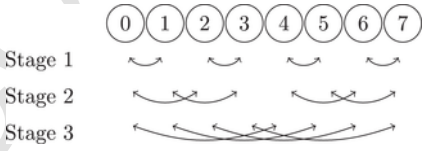


Fig. 1. AllReduce using recursive doubling with pairwise exchange shown using traditional illustration.

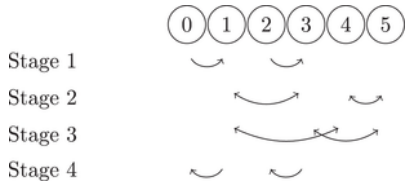
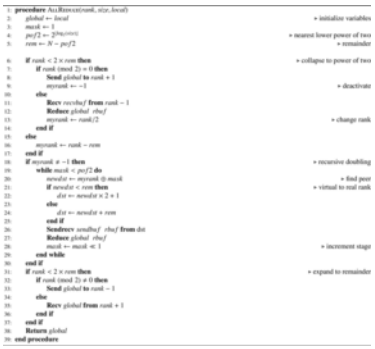


Fig. 2. MPICH fix of non-power-of-two case with six processes using the traditional illustration.



Algorithm 1. Recursive Doubling AllReduce.

rithm is limited by the base of the logarithm which establishes the number of stages which are required. The latency of the network messages also limits recursive doubling since each stage requires each process to exchange messages with a peer process.

Rabenseifner et al. [17] also address the case of a non-power-of-two process count. By using a similar elimination they managed to reduce the rounds required to $\lceil \log_2 N \rceil + 1$ for small messages. By using a particular factorization of the process count they reduce the problem into sets of smaller AllReduce operations.

3. Model

The recursive doubling algorithm presented in Section 2 is based on a hypercube AllReduce algorithm. On a hypercube network, neighbours exchange messages in a pairwise fashion. A simple model such as the postal model with an additional computation term by Chan et al. [7] can be used to model the entire AllReduce.

In the postal model the cost of communication is $\alpha + n\beta$, where α is the network latency and β is the bandwidth. An additional $n\gamma$ term can be used to represent computation if required, where n is the number of bytes which are transmitted and computed. The model assumes a fully connected topology, no network conflicts, and only single message transmissions (alongside other properties that are less important for an AllReduce). Using this model for the recursive doubling with pairwise exchange algorithm results in a total cost of:

$$(\alpha + n\beta + n\gamma) \times \log_2 N \quad \Big| \quad N = 2^k, \quad k \in \mathbb{N} \tag{1}$$

The first term is the cost per stage of the AllReduce while the second term is the number of stages which will be performed. The MPICH fix simply adds two stages to the logarithmic term. Assuming $\beta \ll \alpha$ and $\gamma \ll \alpha$, then for small values of n, both terms tend to be much smaller than the latency. The cost of a small message AllReduce is therefore:

$$\alpha \left(\lfloor \log_2 N \rfloor + \begin{cases} 0 & \text{if } N = 2^k,\ k \in \mathbb{N} \\ 2 & \text{otherwise} \end{cases} \right) \tag{2}$$

From this we can observe that recursive doubling requires improvements in network latency to further improve performance for small messages.

To explore further options, a model that better represents modern hardware is required. Hoefler et al. [13] introduced the LoP model, which modified the popular LogP [8] model to include message pipelining. This functionality was present on the Infiniband network at the time. The LoP model decomposed a message roundtrip into pipelining, processing and saturation terms. This allows representation of observed behaviour for a minimum in roundtrip time for a small number of processes, after which the time increases again. This was used in Hoefler et al. [11] to improve performance of the dissemination barrier.

Using the LoP model was difficult, because the terms of a six dimensional fitting function did not map directly to real world variables. Therefore the LogfP was introduced by Hoefler et al. [12] to address the analytic limitations of LoP while still modelling message pipelining. The LogfP model allows f messages to be sent for free, but above that number the message cost is as in the LogP model. Round trip times were shown to be modelled well by LogfP, but modelling AllReduce operations only requires the local overhead of issuing a message.

We modified the postal model to accommodate message pipelining by removing the original property requiring single message transmission. In addition we decomposed the α term into $\alpha_{?p}$ and $\alpha_{?r}$, where $\alpha_{?p}$ is the part of the latency cost of issuing a send which can be overlapped with subsequent sends, and $\alpha_{?r}$ is the remainder of the latency cost of the sends (which cannot be overlapped). For example, $\alpha_{?r}$ may include critical sections in the MPI library or processing time in the network interface, whereas $\alpha_{?p}$ includes propagation delay time in the network itself. If the number of potentially overlapping messages sent is b, the cost of

sending multiple messages in the pipelining postal model is given by:

$$\alpha_p + b \times \left(\alpha_r + n\beta + n\gamma\right) \tag{3}$$

With this improvement, a model for a recursive multiplying method with an additional requirement for message pipelining capabilities in the underlying network can be constructed. At each stage, instead of exchanging a message with one peer, as in the recursive doubling algorithm, each rank exchanges messages with b peers. If we assume small messages such that the terms involving n are zero, the cost of an AllReduce operation with recursive multiplying is:

$$\left(\alpha_p + b\alpha_r\right) \times \log_{b+1} N \quad \Big| \quad N = (b+1)^k,\ k \in \mathbb{N},\ b \geq 1 \tag{4}$$

Note that this includes the recursive doubling case when $b = 1$.

Fig. 4 shows the cost of the recursive multiplying AllReduce plotted for a range of b and N values with an $\alpha_{?p}$ to $\alpha_{?r}$ ratio of 4. The minimal curve is only dependent on the ratio $\frac{\alpha_p}{\alpha_r}$ and can be expressed using the Lambert W function which is an inverse relation of $ze^{?z}$. The value of b which results in the minimum cost for a given hardware capability is given by:

$$b_{\text{opt}} = e^{W\left(\frac{\alpha_p - \alpha_r}{e\alpha_r}\right)+1} - 1 \tag{5}$$

A ratio equal to one means each message costs the same amount, while a ratio higher means sending more messages is cheaper. Fig. 3 shows $b_{?opt}$ for a range of ratios. An overlap ratio of less than one is not reasonable.

The algorithm as described would require N to be a power of $b + 1$ which is limiting for applicability. However a sequence of b numbers can be used to further extend the algorithm. By allowing separate stages in the algorithm to use different b values, many sizes of AllReduces can be performed.

The time cost of this generalised algorithm is not easily expressible with a logarithmic term, however the per stage cost remains the same as Eq. (4). The number of stages and the size of each stage depends on the decomposition of the AllReduce operation. The decomposition is simply a chosen factorisation of N, which can include non-prime factors to efficiently use the multicast functionality:

$$N = \prod_{i=1}^{s} \left(b_i + 1\right)$$

The number of stages s is determined by the number of factors in the decomposition, while the base of a particular stage is the factor of that stage. Using this decomposition of N we can express the total cost of all stages as a summation of our model

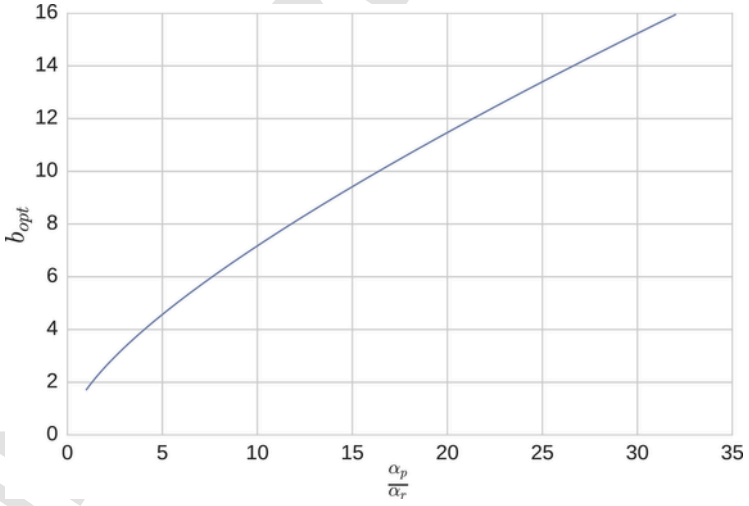


Fig. 3. Optimal b value for given message pipelining ratio given by Eq. (5).

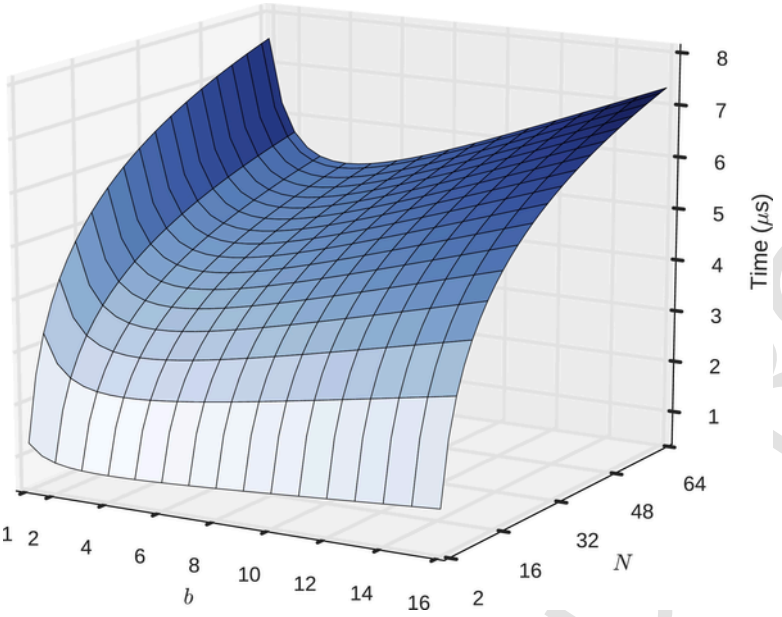


Fig. 4. Execution time of AllReduce as a function of b and N, as modelled in Eq. (4), with $\alpha_p = 1\ \mu s$ and $\alpha_r = 0.25\ \mu s$.

cost shown in Eq. (3). The total cost within the model for an AllReduce operation is:

$$\sum_{i=1}^{s} \left(\alpha_p + b_i \alpha_r\right) = \alpha_p s + \alpha_r \sum_{i=1}^{s} b_i$$

4. Algorithm

4.1. Design

The recursive multiplying method is a generalization of recursive doubling and relies on the ability to multicast messages. As shown in Section 3 the multicast capability gives us the ability to reduce the number of stages that the algorithm needs to perform. This requires factoring the number of processes N.

Several limitations exist to this algorithm other than the requirement for message pipelining. The potential for the prime factorization to include large prime numbers, which are inefficient to multicast, is a problem which can be addressed in a similar manner to the fix utilized in MPICH. In addition, having several messages arrive per stage requires all participating processes to have more memory available.

The specific amount of memory required is given by the schedule. In comparison with recursive doubling which requires one buffer per stage, recursive multiplying requires $b_i - 1$ buffers per stage. As with the memory requirement, the total amount of data communicated at each stage is also increased. Due to this, the network needs to provide enough bandwidth to handle multiple messages, and be able to handle potential congestion.

The prime factorisation of N can be used to conveniently find an acceptable schedule, which can be performed through recursive multiplying. However, the prime factorisation will likely provide many small factors which are well below the multicast ability. To reduce the number of stages and maximise the usage of multicast, these smaller stages are combined into larger combinations. This can be done exhaustively to find all possible schedules which would result in a correct result. To find the minimal time schedule it is possible to measure the $\alpha_{?p}$ and $\alpha_{?r}$ values and then use the models discussed in Section 3 to predict the required time.

Fig. 5 shows a schedule performed with the same setup as Fig. 2. By using a factor, other than two, it is possible to perform an AllReduce of six processes within two stages, compared to the recursive doubling four stages. Equally for an AllReduce size of ten a potential schedule would be (2,5) compared to the five stages required by recursive doubling.

In the case of large primes when it is inefficient to multicast a generalisation of the fix used previously is available. Utilising the message pipelining ability to perform cheap multicast we collapse groups of processes in a subgroup. Compared to the previous fix this allows a larger reduction in group size for the remaining execution of the AllReduce. This generalisation still re-

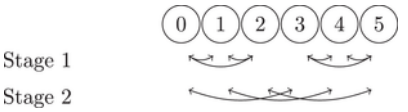


Fig. 5. AllReduce operation with six processes using traditional illustration.

quires an additional stage before and after the execution of the internal AllReduce to allow for the processes which did not participate directly to receive the final result.

It is possible to visualise the recursive multiplying algorithm as a recursive AllReduce algorithm as shown in Fig. 6. The recursive doubling algorithm represents each stage as an exchange on a single dimension, compared to the recursive multiplying algorithm which has multiple processes exchange partial results within a single stage. Then all groups in a stage communicate with their respective peers in further stages. This visualisation uses n-dimensional hyper-cuboids compared to binary hypercubes for the recursive doubling case. An n-dimensional hypercuboid can be used with a large AllReduce operation.

Another approach to dealing with large primes is to do merging, which does not require two additional stages. This allows a composite number to be used instead of a multiple of a base. The first stage is executed performing the first stage of the schedule while the exposed remainder process broadcasts its own value to all processes as required. During the final stage all processes of a group send their final value to the remaining processes which then reduce these themselves. By decomposing the size of the AllReduce operation into two numbers, one of which is well factorisable, we can make efficient use of multicast.

In the given example shown in Fig. 7 only two stages exist to perform an AllReduce across seven processes. The first stage consists of the first six processes performing an AllReduce within two groups, while the seventh process broadcasts its value to an entire group at the same time. This allows all members of that group to calculate the reduction with the contribution of the seventh process. During the second, and final, stage three groups of two processes perform a pairwise exchange, while a single group also sends its partial results to the seventh process to reduce them by itself. By using prime merging we enabled the seventh process to receive all required information within the two stages instead of the four required by the generalised fix.

4.2. Implementation

The pseudocode for recursive multiplying is presented in Algorithm 2. The pseudocode shows several required generalisations, compared to the recursive doubling in which simplifications of these were enough. The transformation from virtual ranks to real ranks is done similarly with a branching statement, though the transformation is more complex than the power-of-two-case. In addition, the masking to find the relevant peers for a stage has changed from an exclusive-or operation, to be a group and offset calculation. Finally, the mask incrementing has changed from a left shift operation, to a multiply by the stage base.

The schedule is passed directly to the AllReduce operation globally and not computed on the fly. This enables library implementers to have control over which schedules are used when. The schedules consist of instructions which some stages will execute depending on previous stages. The general factor type is a simple "a*B*" stages where *B* is the base for that stage.

The collapse and expand instructions are encoded as either "c*T*m*B*" or "e*T*m*B*". The threshold *T* is defined to be divisible by *B* and is used as the delimiter between the collapsing and shifting peers. *B* is the base which is used during the collapse and expand stages. The pseudocode for the collapse and expand phases are shown in Algorithms 3 and 4.

For the merging method the merge is encoded as "m*R*g*G*a*B*" and the inverse of merge as "n*R*g*G*a*B*". The *R* is the remaining processes which are to be merged into the internal decomposition. The *G* and *B* describe the face of the hypercuboid for which the merging is used. The pseudocode for the merging method is shown in Algorithms 5 and 6.

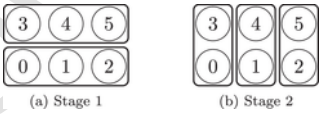


Fig. 6. Hypercuboid view of AllReduce operation with six processes.

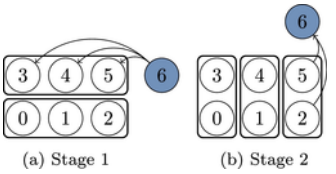


Fig. 7. Hypercuboid view of prime merging with seven processes.

Algorithm 2 Recursive Multiplying AllReduce

```
1: procedure AllReduce(rank, com, schedule)
2:     value ← com                                              ▷ initialize variables
3:     stage_mask ← 1
4:     pthres ← 0
5:     pbase ← 1
6:     wid ← rank

7:     for stage in schedule do
8:         if type(stage) is factor then
9:             sfactor ← factor                                 ▷ recursive multiplying
10:            sbase ← sfactor × stage_mask
11:            if wid ≠ −1 then
12:                for index ∈ [0, sfactor−1) do                ▷ find peers
13:                    mask ← (index +1) × stage_mask
14:                    block ← ⌊wid/sbase⌋ ×sbase
15:                    offset ← (wid + mask) mod sbase
16:                    peer ← block + offset
                                                                ▷ virtual to real rank
17:                    if rpeer < pthres then
18:                        rpeer ← peer × pbase + pbase −1
19:                    else
20:                        rpeer ← peer + pthres/pbase × (pbase −1)
21:                    end if
22:                    Send non-blocking value to rpeer
23:                end for
                                                                ▷ complete stage
24:                for peer ∈ [0, sfactor) do
25:                    Recv value from peer
26:                    Reduce value rbuf
27:                end for
28:                Wait on sends
29:            end if
30:            stage_mask ← stage_mask × sfactor
```

Algorithm 2.

Algorithm 3 Recursive Multiplying Collapse

```
31:        else if type(stage) is collapse then
32:            pthres, pbase ← collapse
33:            if rank < pthres then
34:                if rank  (mod pbase) ≠ (base−1) then
35:                    peer ← ⌊rank/pbase⌋ × pbase + pbase −1
36:                    Send value to peer
37:                    wid ← −1
38:                else
39:                    Recv rbuf from peer
40:                    Reduce value rbuf
41:                    wid ← ⌊rank/pbase⌋
42:                end if
43:            else
44:                wid ← rank − pthres/pbase × base −1
45:            end if
```

Algorithm 3.

5. Results

All the experiments reported in this Section were run on the ARCHER [1] supercomputer, a Cray XC30 machine with 4920 compute nodes, each with two 12-core Intel E5-2697 v2 CPUs. The interconnect is the Cray Aries in a Dragonfly topology. The environment used was:

- "PrgEnv-cray/5.2.56"

Algorithm 4 Recursive Multiplying Expansion

```
46:     else if type(stage) is expand then
47:         if rank < pthres then
48:             if rank (mod pbase) = (base−1) then
49:                 for b do
50:                     peer ← wid × pbase + b
51:                     Send non-blocking value to peer
52:                 end for
53:             else
54:                 Recv value from peer
55:             end if
56:             Wait on sends
57:         end if
```

Algorithm 4.

Algorithm 5 Recursive Multiplying Merge

```
58:     else if type(stage) is merge then
59:         remainder, groups, factor ← merge
60:         group_size ← factor × stage_mask
61:         if rank < remainder then
62:             wid ← −1 − rank
63:             group ← rank (mod groups)
64:             group_first ← remainder + group × group_size
65:             for each process in group do
66:                 peer ← group_first + idx
67:                 Send non-blocking value to peer
68:             end for
69:         else
70:             wid ← rank − remainder
71:             group ← ⌊wid / group_size⌋ × group_size
72:             for each peer in group do
73:                 Send non-blocking value to peer
74:             end for
75:             Wait for all receives
76:             Reduce all buffers
77:         end if
78:         Wait on sends
79:         stage_mask ← stage_mask × factor
```

Algorithm 5.

- "dmapp/7.0.1-1.0502.10246.8.47.ari"
- "cray-mpich/7.2.6"
- "pmi/5.0.7-1.0000.10678.155.25.ari"
- "ugni/6.0–1.0502.10245.9.9.ari"

In all cases we used one rank per node, so that all communication is over the network and not in shared memory on the node. All measurements are performed using an AllReduce summation operation of a single 8 byte integer.

5.1. Multicast performance

The pipelining postal model presented in Section 3 is interesting in the context of AllReduce since no round trip occurs within the algorithm. Only the local overhead of issuing a message and the latency until a message from a different peer arrives is important.

The benchmark used to measure both the $\alpha_{?p}$ and $\alpha_{?r}$ values is very similar to the PingPong benchmark used by Hoefler et al. [13]. We replicate the function of a multicast operation with multiple pipelining messages. The timing routines are issued directly before and after the message initiation. This explicitly measures the overhead in the LogP/LoP/LogfP models.

Fig. 8 shows the distribution of timing results of a multicast. The number of peers shown is the number of messages which were sent in a single operation. The centre of the notch is the median value, while the width of the notch represents the confidence interval of the median. The box extents are the 25th and 75th percentile of the measured results. The error bars are shown

Algorithm 6 Recursive Multiplying Inverse Merge

```
80:         else if type(stage) is invmerge then
81:             remainder, groups, factor ← invmerge
82:             groupsize ← factor × stage_mask
83:             if rank < remainder then
84:                 Wait for all receives
85:                 Reduce received buffers
86:             else
87:                 group ← ⌊wid / group_size⌋ × group_size
88:                 for each peer in group do
89:                     Send non-blocking value to peer
90:                 end for
91:                 for each remainder do
92:                     if remainder (mod groups) = wid (mod groups) then
93:                         Send non-blocking value to remainder
94:                     end if
95:                 end for
96:                 Reduce all buffers
97:                 Wait for sends
98:             end if
99:         end if
100:    end for
101:    Return value
102: end procedure
```

Algorithm 6.

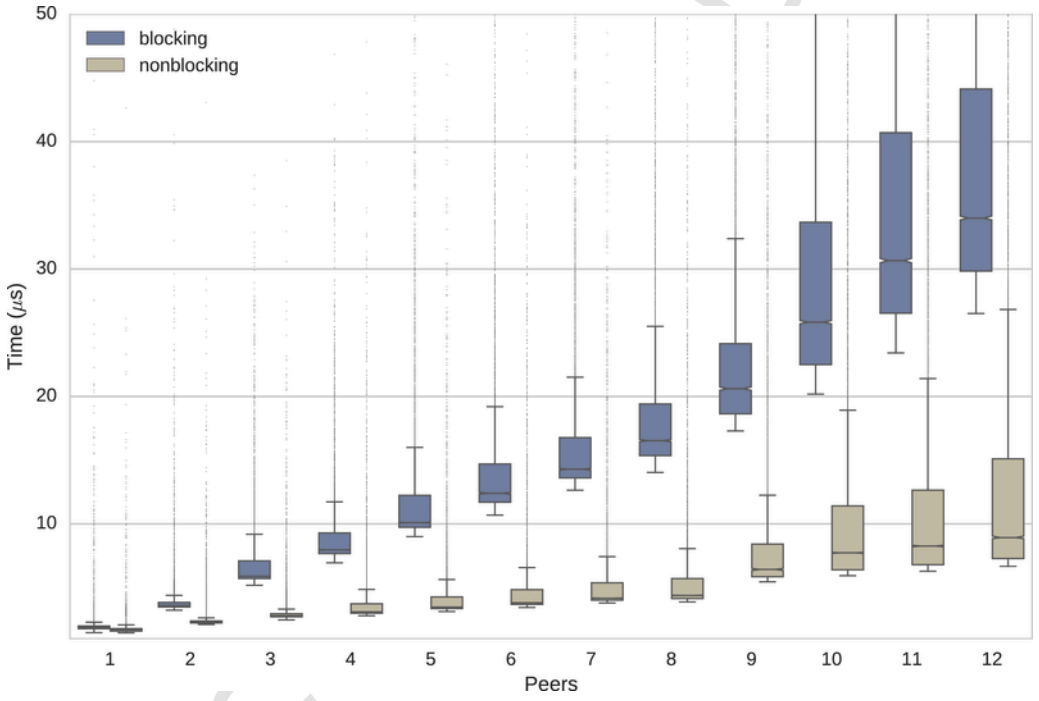


Fig. 8. Distribution of times with varying multicast size with blocking and non-blocking sends.

as either a minimum or maximum of the measured values if the value is within 1.5 times the interquartile range, otherwise they are 1.5 times the interquartile range. Outliers are shown above the whiskers by dots. Since these results were gathered on a live system much noise is encountered, but the underlying distribution is clearly positive skewed. A fixed minimum is expected since the message transmission is limited by the speed of light. The tail of the distribution is not bounded and could be very large.

The multicast was implemented in two different ways, using blocking and non-blocking sends. A multicast operation constructed with non-blocking sends clearly outperforms blocking sends. An interesting artefact occurs at eight peers when both blocking and non-blocking have a slight discontinuity which does not occur with any higher peer count. This could be due to packet combining, since the payload size is 8 bytes and the packet size is 64 bytes.

A linear regression is performed using minimum and median values to approximate the $\alpha_{?p}$ and $\alpha_{?r}$ values seen experimentally. The values are presented in Table 1 using only the experimental results from one to eight peers.

5.2. Allreduce benchmark

The benchmark to evaluate the algorithm presented in Section 4 is implemented using the Cray DMAPP library [4], which supports a PGAS-based approach to communication. Although the algorithm presented does not explicitly require single-sided communication, using Cray DMAPP allows the least amount of time between message issues without a large software stack, which enables us to maximize the message pipelining.

The memory consumption is less efficient than a point-to-point channel implementation. Both approaches are difficult to quantify: point-to-point channels consume $O(1)$ memory, but $O(\log_{?2} N)$ channels exist in memory. The PGAS-based implementation utilises a memory array allocated in the data segment of the application which stores the addresses to write to for each peer.

The live ARCHER system was used for measurements, therefore noise is present throughout all results. The experimental setup is to measure all results in blocks of 10 AllReduce operations. This is done to ensure a higher resolution of the timing routines and to reduce or average skew effects present in the measurements. The number of blocks is equal for each node allocation and set at 250. This is to ensure a large number of samples on different node allocations, but work within budget constraints.

The node allocations given on the live ARCHER system are variable and dependent on job requirements. Therefore at least forty node allocations were taken, then an evaluation of the results was performed and more node allocations were measured if the change in median and mean was not below 5%. This allows measurement of all potential noise sources such as hardware failures, OS noise, network noise and system load.

5.3. Allreduce schedule comparison

We compare the performance between an implementation of recursive doubling and an appropriate schedule for a given collective size. The schedules used to represent recursive doubling were exactly the behaviour which would be performed in MPICH. The power-of-two cases were handled by a series of "a2" stages. The non-power-of-two stages were handled by collapse and expand stages with a series of "a2" stages between them.

The results for recursive multiplying used the schedules presented in Table 2. Results for both the improvement on the minimum and median are shown, with decreases in execution time ranging from 8% and maximum improvements of 40%. The number of blocks measured for each process count is shown. All process counts were measured with at least 100,000 AllReduce operations due to the large variance on ARCHER.

The schedule choice was done experimentally by exhaustively measuring all schedules possible for a given size and then selecting the schedule with the minimal value of the median execution. This method of choosing which schedule to use is not representative of what would be done for each AllReduce execution. In production the machine administrator would execute a benchmark which would evaluate all schedules and then statically assign this for a certain size.

The performance results of the benchmark are presented in Fig. 9. The errors are shown as described for Fig. 8. As can be seen, the minimum values are significantly less than the median values, with considerable spread of all measurements. Neither

Table 1
$\alpha_{?p}$ and $\alpha_{?r}$ values from multicast experiment (non-blocking sends).

	Minimum (μ s)	Median (μ s)
$\alpha_{?p}$	1.34	1.51
$\alpha_{?r}$	0.34	0.38

Table 2
Percentage decrease in minimum and median execution times from recursive doubling to recursive multiplying.

Process count	Schedule	Blocks	Min RD/RM (μs)	Min %	Median RD/RM (μs)	Median %
4	a4	10,500	2.36 / 1.87	21.1	4.55 / 3.65	19.7
6	a6	10,250	4.76 / 2.87	40.0	8.72 / 5.75	34.1
8	a2,a4	10,750	4.37 / 3.54	18.9	9.38 / 7.23	23.0
12	a3,a4	10,000	7.03 / 4.43	37.0	15.3 / 11.6	24.4
16	a4,a4	12,500	6.85 / 4.98	27.3	19.1 / 13.8	27.6
24	a4,a6	13,750	9.99 / 6.91	30.8	29.7 / 25.5	14.2
32	a8,a4	10,000	12.9 / 8.95	30.8	30.9 / 25.2	18.4
48	a8,a6	10,000	19.8 / 13.3	33.2	38.7 / 32.1	17.1
64	a8,a8	10,000	24.2 / 16.5	31.9	47.4 / 39.9	15.9
96	a8,a3,a4	12,500	25.3 / 20.7	18.1	51.7 / 47.6	7.96
128	a8,a4,a4	10,000	25.1 / 17.9	28.9	101.0 / 88.4	12.5

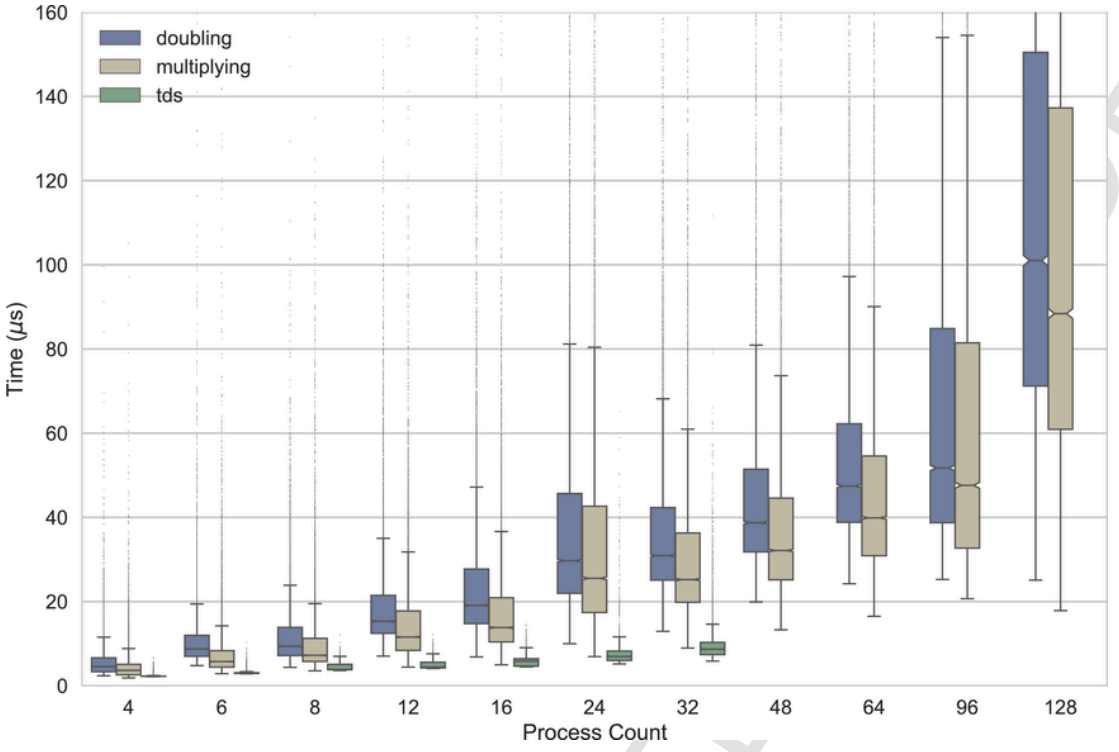


Fig. 9. Execution times for varying sizes of AllReduce using both recursive doubling and recursive multiplying.

the minimum or median results follow a logarithmic curve when going to large scales. Comparing recursive doubling to the best recursive multiplying schedule there is a significant advantage by using message pipelining: the median value for recursive multiplying is near the 25th percentile value for recursive doubling. Important to note is the reduction in improvement as the scale at which the AllReduce is performed grows, but it is important to note that with $N = 128$ the gains have improved.

The *tds* results shown in Fig. 9 are executions of the respective schedules of recursive multiplying on a separate Cray XC30 machine used for testing and development of the ARCHER supercomputer. We used this cluster to understand what impact the congestion present on the large computer has on the execution of the algorithm. From the results, the congestion is the most contributing factor of the executions times.

5.4. Message size scalability

The recursive multiplying algorithm was designed to improve the latency of small sized messages. To test how capable the algorithm is of accepting larger messages we performed a message size sweep on both 8 and 64 process count executions. A subset of all possible schedules was chosen to be evaluated, which included the recursive doubling schedule. The schedules used are shown in the legend of the respective Figs. 10 and 11. The message count is the count given if an MPI function call were executed. All messages are multiples of 8 bytes, for example message count 8 corresponds to 64 bytes.

Fig. 10 shows an expected behaviour with a latency bound region and then a switch into a bandwidth bound region at a message count of approximately 24–32. The (a2,a4) schedule performs surprisingly well with the minimum execution time being the best one for the entire sweep and the median being approximately equivalent to recursive doubling at higher message counts.

Fig. 11 shows the message size sweep results for $N = 64$ executions. Due to running this benchmark later compared to previous results the environment of the ARCHER supercomputer has changed enough to see a strong difference in execution times, however we are comparing only the data shown on the plot. As seen at the low end of the message count axis recursive doubling is likely the best option, but as the message count increases the performance of recursive doubling is significantly worse than at the low end. At 512 message count(4096B) the recursive multiplying schedules clearly outperform the recursive doubling schedule. We cannot explain this result since recursive multiplying was designed to perform well with small messages. We suspect that the adaptive network allows more bandwidth to be used since the algorithm is sending many more messages for each stage and therefore puts more network load on surrounding paths.

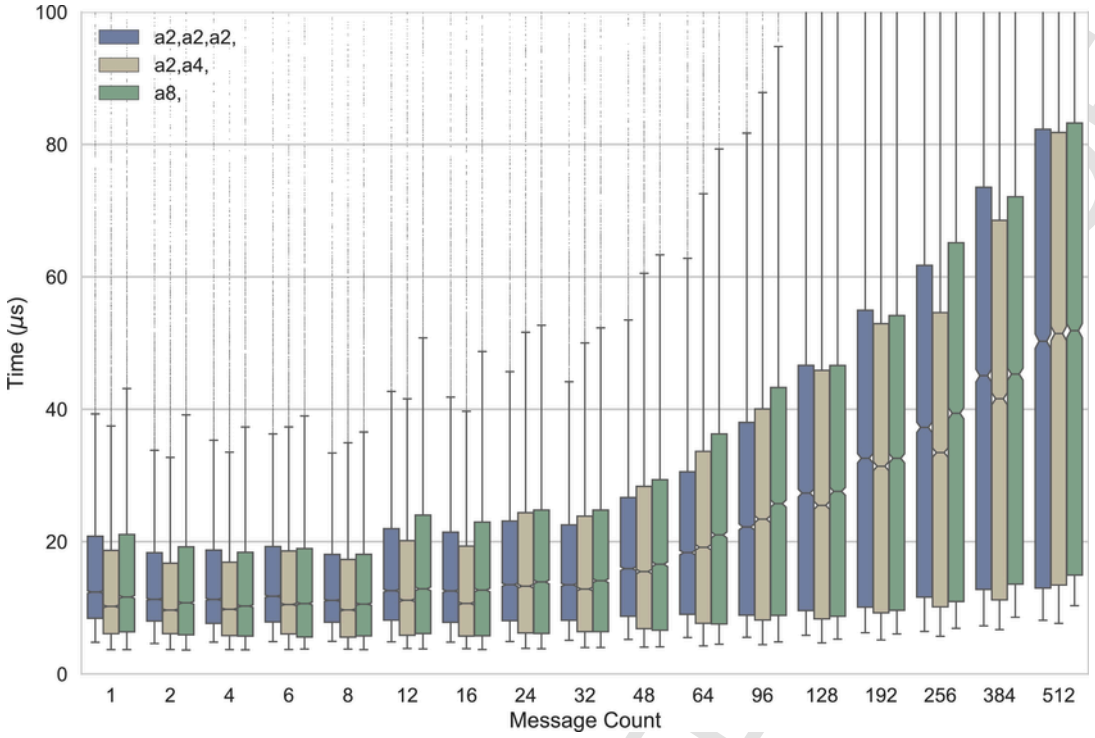


Fig. 10. Executions times for varying message sizes using eight processes with all possible schedules for $N = 8$.

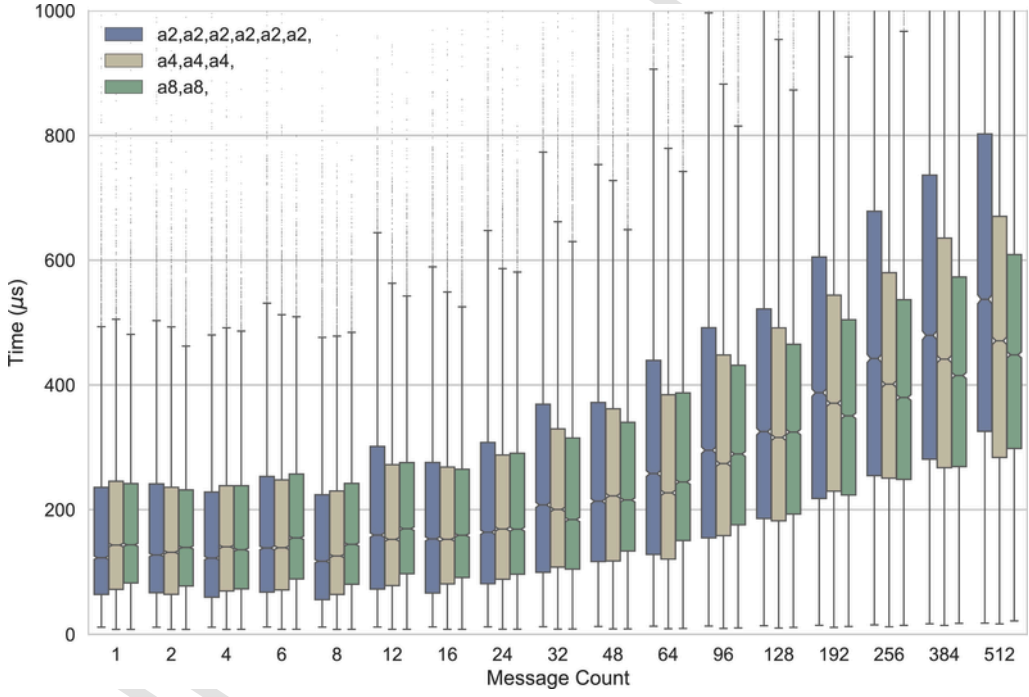


Fig. 11. Executions times for varying message sizes using 64 processes with a subset of schedules chosen.

5.5. Allreduce model comparison

The model presented in Section 3 can be used to predict the time required for an AllReduce operation. To measure the accuracy of this approach we used the prediction of the model compared to the experimental results given in Section 5.3. Using the values for $\alpha_{?p}$ and $\alpha_{?r}$ evaluated previously for the median and minimum we can use the model to predict the minimum and median values for AllReduce operations.

Fig. 12 shows the relative difference of the experimental results to the predictions by the model. As can be seen, the minimum values follow the model well until 24 nodes is reached. An upwards trend is clear from there onwards similarly to the median values. The median values show a consistent increase in the difference. The difference between the theoretical and actual values is likely due to skew of process arrival times. This increases as the size of the AllReduce increases, since it is more likely that a process is delayed.

5.6. Block size systematic error analysis

The block size chosen in Section 5.2 was chosen such that the noise encountered from the measurements did not increase the maximum value, but also to ensure a high resolution for the timing routines. Fig. 13 shows resulting distributions of a 64 process AllReduce using the best schedule, presented in Table 1, with varying number of samples per block of measurements. The red lines show the median(upper) and minimum(lower) of the block size of ten for easier comparison.

As seen in Fig. 13 the minimums of all measurement block sizes are consistent. Therefore we have confidence the minimums were captured with the results presented in Fig. 9. The medians show a slight upward trend correlated with block size. This causes the median calculated using a block size of ten to overestimate by approximately ten microseconds compared to the median using a block size of one. This overestimation is true for both the recursive doubling and the best recursive multiplying schedules, therefore the relative difference as discussed in Table 1 is valid.

5.7. Experimental-model correlation

As seen in Fig. 12 the prediction using the model is not reliable for either the minimum or the median for the best schedule presented for the process counts. Using the model for prediction of the best schedule to use is not clear from the analysis. We plot all evaluated factored schedules in Figs. 14 and 15 using the experimental runtime value and the model prediction for that schedule. The AllReduce size is shown via the color of the points.

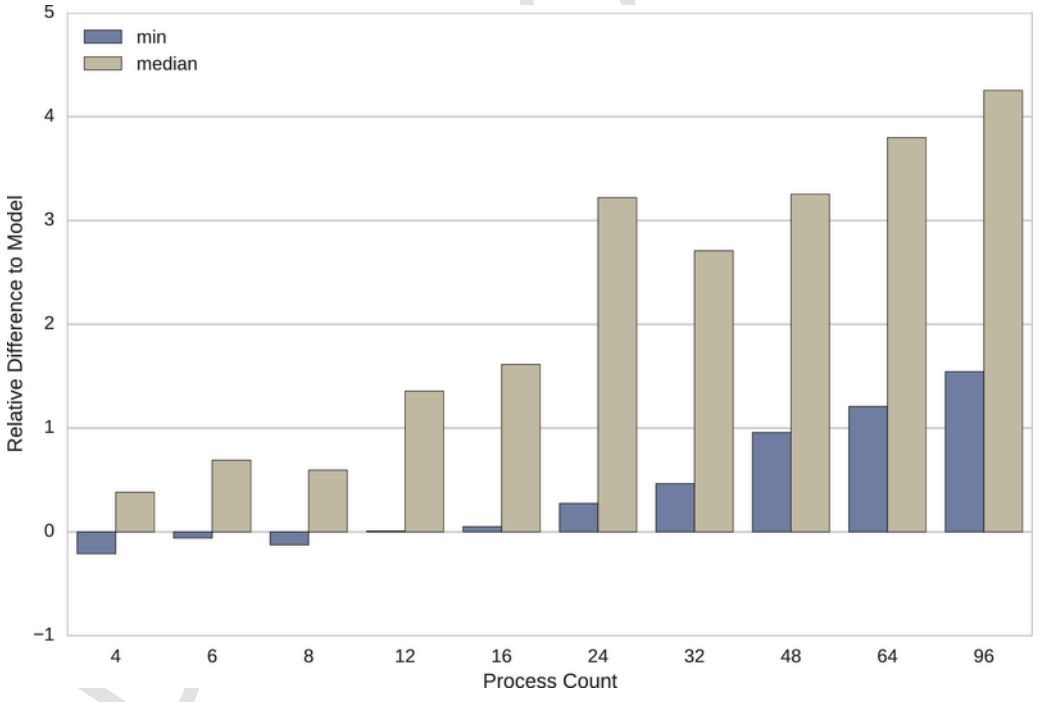


Fig. 12. Relative difference of experimental minimum and median to prediction from pipelining postal model.

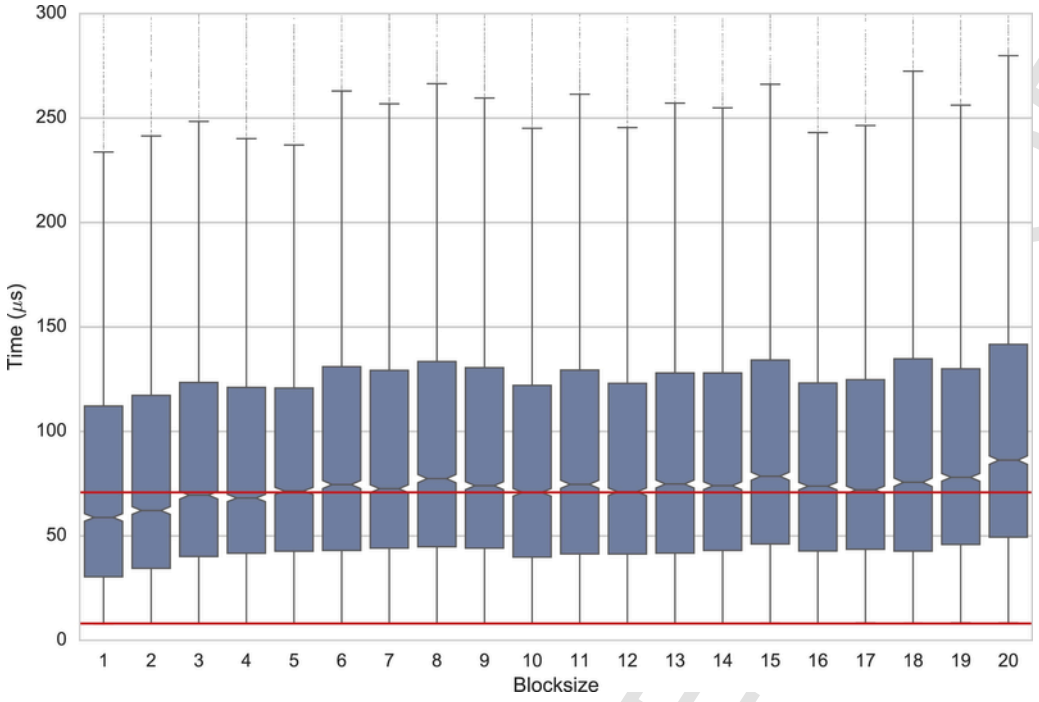


Fig. 13. Distributions using different block sizes for the (a8, a8) schedule with 64 processes. The red lines show the median and minimum of the block size 10 distribution for comparison. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

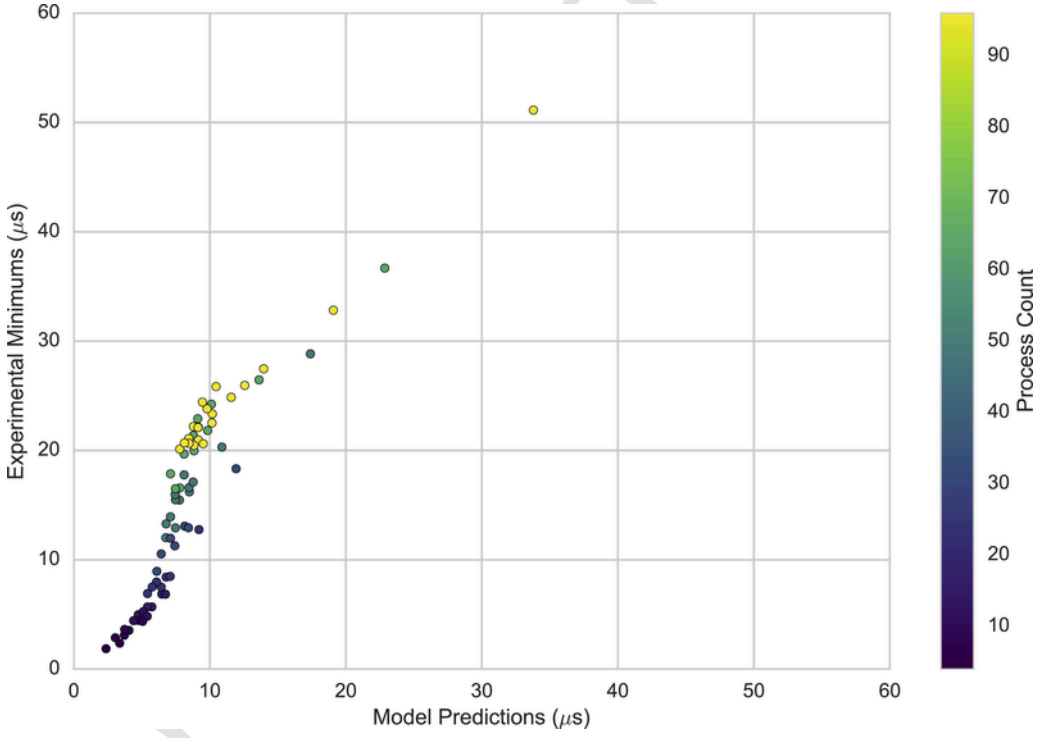


Fig. 14. Correction plot for experimental and model results using minimums.

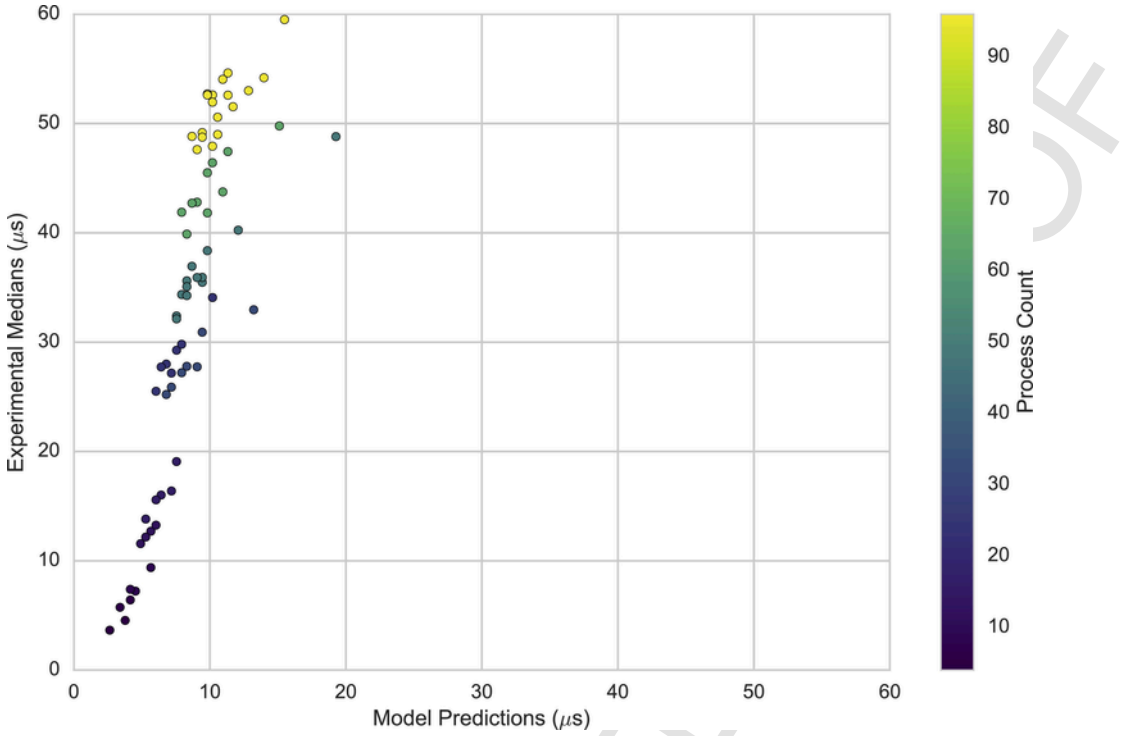


Fig. 15. Correlation plot for experimental and model results using medians.

As shown in Fig. 14 the schedule runtimes are not correctly predicted, but the clustering indicates that schedules are correctly identified as good or bad choices for a certain AllReduce size. Fig. 15 shows that the median runtime is also incorrectly predicted, but the correct schedule would also be chosen. The error in the median prediction is evident from the plot.

These results show that neither the minimum nor the median are predicted well by the theoretical model, but that a good schedule would be chosen regardless due to the clustering. This ill fitting is likely due to the effect that congestion has on the experimental results, which as discussed in Section 5.3, is a large factor in the performance of the algorithms. The model does not attempt to model congestion and therefore fails to predict correct times. The model does approximately match the results shown in Fig. 9 for the *TDS* system.

5.8. Cray MPI comparison

A direct comparison to the MPICH implementation of AllReduce is not representative due to the lack of an MPI library above our benchmark, but it is included to illustrate the benefits. Fig. 16 shows the MPICH result alongside the recursive doubling and best recursive multiplying schedule. As shown the minimums of the best schedule outperform both MPICH and the recursive doubling schedule as expected. This is true for the median values also.

From Fig. 16 we can conclude that the Cray MPI implementation is indeed using the recursive doubling algorithm discussed in Section 2. In addition we can see that the library overhead is negligible compared to the cost of the algorithm and congestion since the *mpich* and *rd* results are mostly identical. Finally, we can say that the recursive multiplying algorithm definitely outperforms the MPI_AllReduce implementation even if the library overhead was included.

6. Simulation

6.1. Simulator architecture

The custom built simulator is used to bridge the gap between the experimental results given in Section 5 and the theoretical predictions discussed in Section 3. The simulator is used to analyse and visualize the behaviour of a given algorithm or schedule for recursive multiplying. For this purpose the simulator had to be simple, flexible and moderately performant. The performance was not the priority since only short executions of the algorithms would be simulated, not entire simulations of applications.

This was achieved by using a design loosely based on LogGOPSim [14] written in Python which allows for flexibility. The simulator is structured around the concept for a model which is instantiated as a machine object. The machine is given a pro-

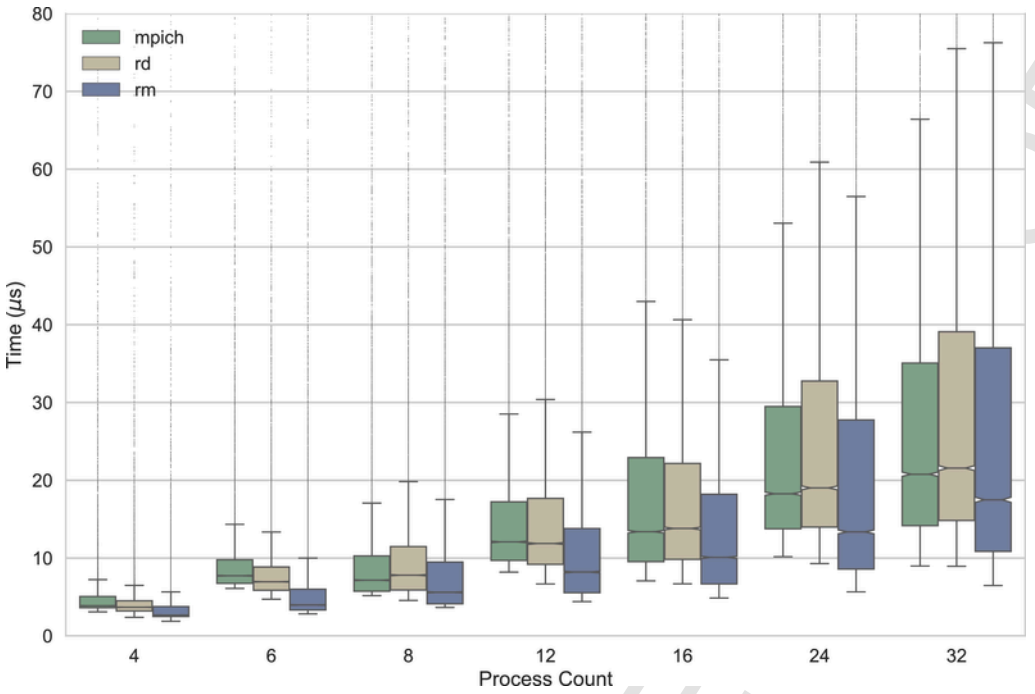


Fig. 16. Execution times comparing directly the MPICH MPI_AllReduce implementation, the recursive doubling schedule as implemented in our benchmark and the best recursive multiplying schedule found.

gram object which it simulates according to the model. This allows for many models to be supported, currently we support the postal model, the pipelining postal model and the LogP model. Once the simulation completes properties of the final state of the machine can be measured using the object. In addition the model can inject properties of the machine such as noise or topology effects. These additional features are currently only partially supported.

The generators were created for usage specifically for recursive multiplying. Given an encoded schedule as discussed in Section 4 the generator can translate the schedule to a program object. In addition to program generation the generator can create all possible schedules for a given process count using factored, split and merged schedules.

The programs are encoded as a directed acyclic graph which at each node have a task associated as metadata. This allows walking the graph by the model implementation of the machine and thereby execute the control flow. The tasks are generic operations which are within the bounds of the models which are executing them. An illustration using the following task types is Shown in Fig. 17:

StartTask The StartTask task type is a helper task for the simulator to easily find entrant nodes in the program graph. This also allows for simple representation and segmentation between separate executions within the same model instance. The task type when encountered in the simulator does not require any simulation time to execute.
ProxyTask The ProxyTask task type is similar to the StartTask type, because it does not require any simulation time to execute. It acts as a link between actual simulation task types. This allows for simplified algorithmic generation of the schedules through the program generator.
SleepTask The SleepTask is provided as a way to block a process in the model from executing until a certain time is elapsed. This can be used to probe skew in program execution.
ComputeTask The ComputeTask task type is given to simulate a given local computation, such as a compute phase in an application. The ComputeTask is the same as the SleepTask since it blocks the process for a certain amount of time, however it is helpful to be able to label tasks.
PutTask The PutTask task type is the most important task as it describes the abstract operation of a put communication between processes. Depending on the model this task type acts differently since not all network models implement a put identically.

6.2. Recursive multiplying analysis

The recursive multiplying algorithm was previously explored both theoretically and experimentally, however both failed to give a good visual representation of the actual execution. The theoretical diagram compresses the overlapping messages to a stage which causes skewing to be hidden while the experimental measurements contain too much noise.

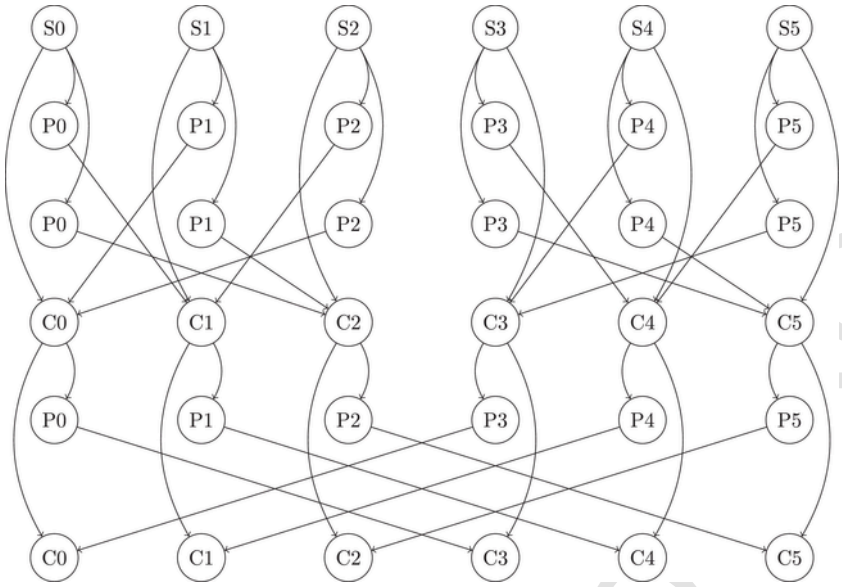


Fig. 17. Program diagram of 6-way AllReduce using schedule (a3,a2). Start tasks are labelled with S, Put tasks are labelled with P and Compute tasks are labelled with C.

The examples shown in the following sections use a simulated pipelining postal model instance with the latency set to 500 ns, the pipeline latency to 100 ns and the bandwidth term set to 0.4 ns per byte. The compute tasks are set to take ten nanoseconds. Since the local computation is a minor cost it is not important.

Fig. 18 shows execution of the recursive doubling schedule. As seen overlapping is not occurring in this case since each sending process is waiting for the message to arrive on the receiving process. In comparison Fig. 19 shows the extreme case of all messages pipelining within a single stage. As shown previously the message pipelining causes the multicast stage to be executed significantly faster.

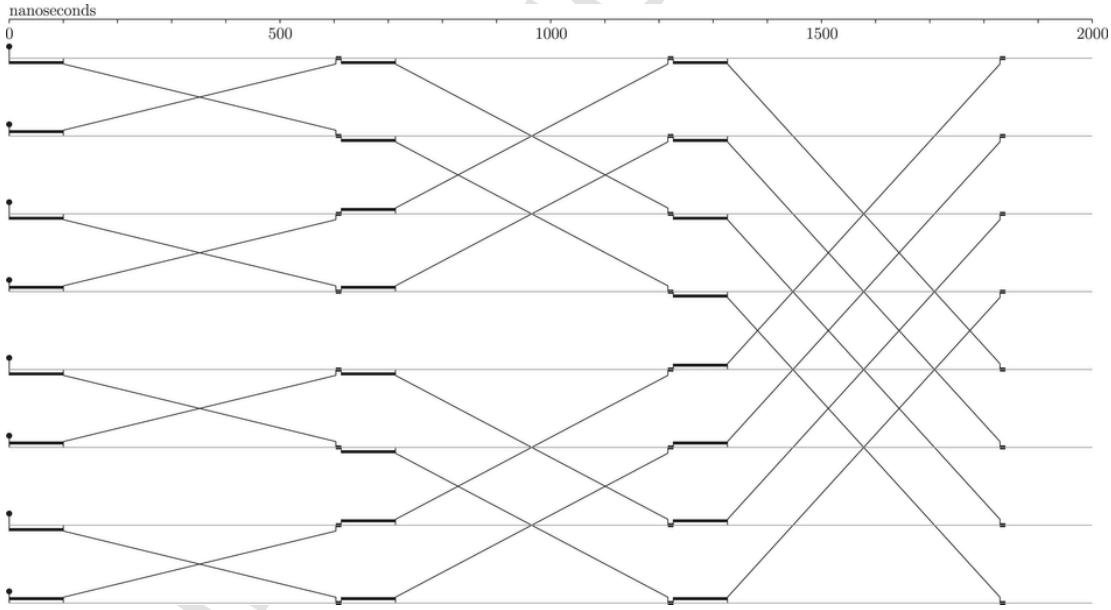


Fig. 18. Simulated execution of the a2,a2,a2 schedule AllReduce across eight processes.

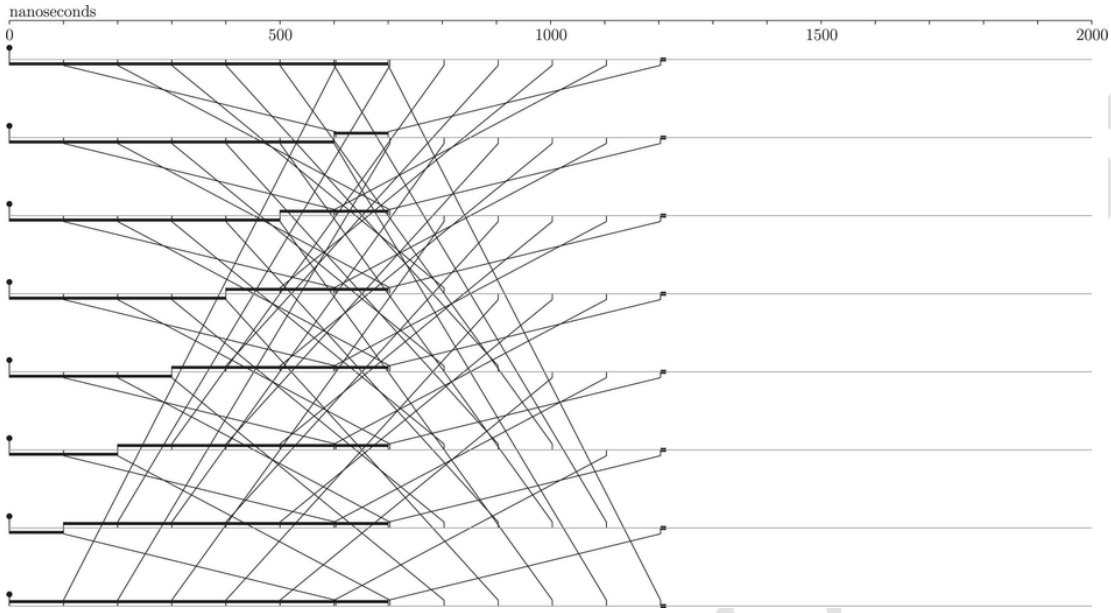


Fig. 19. Simulated execution of the a8 schedule AllReduce operation across eight processes.

6.3. Splitting & merging

The recursive multiplying algorithm can evaluate AllReduce operations across many process counts, however similarly to recursive doubling some process counts are only possible inefficiently. For recursive multiplying these are all prime numbers above a threshold determined by the overlap ratio discussed in Section 3.

The generalised fix method used in MPICH was presented in Section 2 we refer to as a splitting method. Fig. 20 shows the splitting method applied to a $N = 7$ AllReduce. The interesting feature seen with the simulator is the skew which is introduced in an ideal execution of the algorithm. The collapsing processes send their data to their respective peers as required while the two internal peers are already executing stage two of the algorithm. This causes a skewed arrival at the second stage and subsequent stages. As seen the finishing times for each process vary by approximately 700 nanoseconds. The specific skew introduced is dependent on the schedule chosen.

The merging method introduced in Section 4 is shown in Figs. 21 and 22. As seen in the figures the runtime is decreased by approximately one microsecond compared to the recursive doubling schedule in Fig. 20. Similar to the recursive doubling skew is introduced, however the skew is within 100 ns for the first schedule and 200 ns for the second schedule.

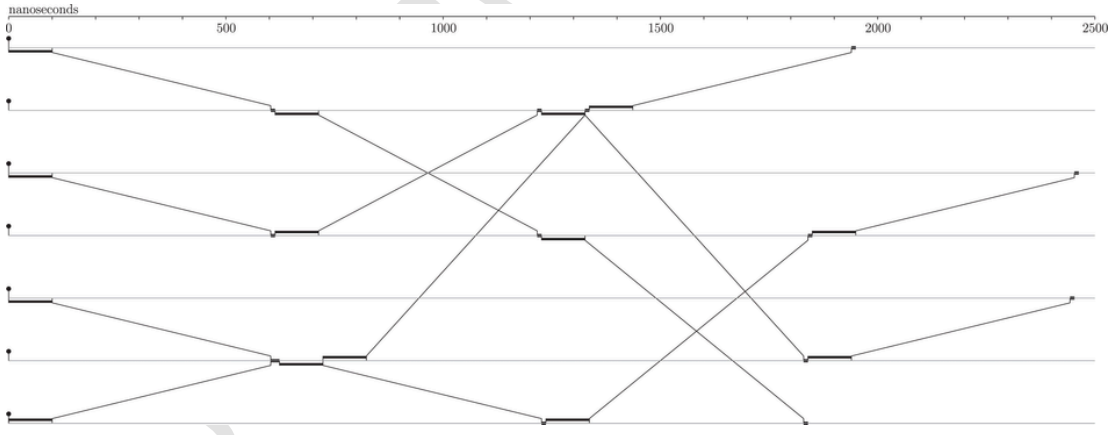


Fig. 20. Simulator timeline of the splitting schedule (c6m2,a2,a2,e6m2) across seven processes.

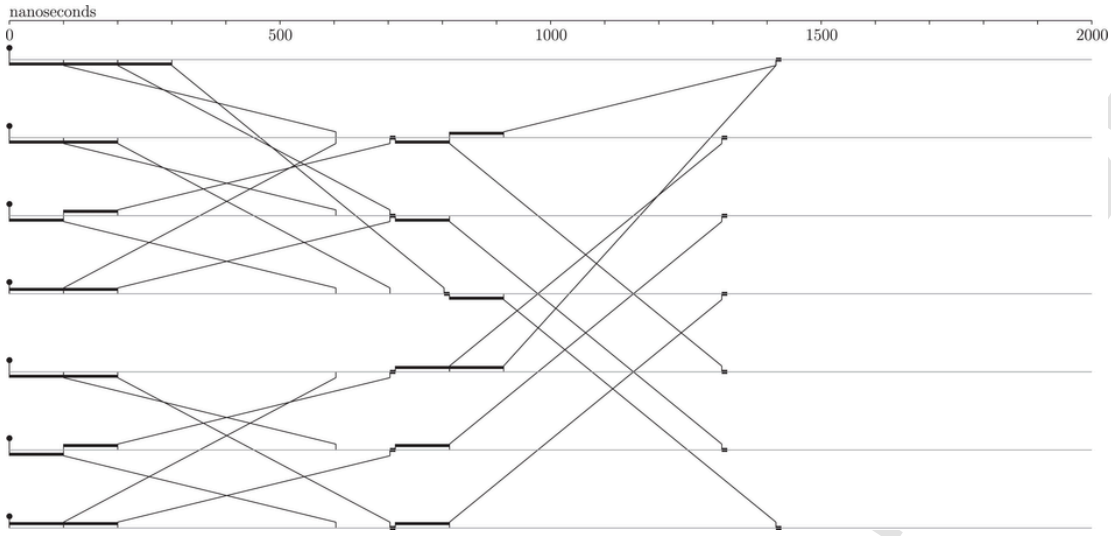


Fig. 21. Simulator timeline of the merging schedule (m1g2a3,n1g3a2) across seven processes.

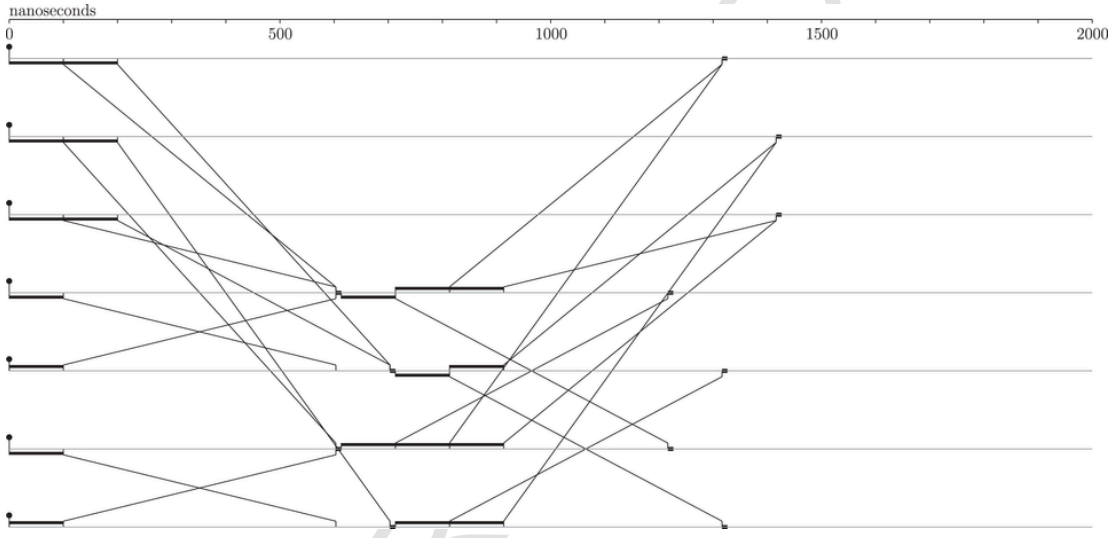


Fig. 22. Simulator timeline of the merging schedule (m3g2a2,n3g2a2) across seven processes.

6.4. 3-2 & 2-1 Elimination

Using the simulator we are able to simulate not only the recursive multiplying algorithm, but also the 3-2 & 2-1 elimination method discussed by Rabenseifner et al. [17]. Fig. 23 shows an AllReduce operation executed using a 3-2 elimination. The simulator shows how the 3-2 elimination can overlap with in time with the pairwise exchange and thereby allow for the $\lceil \log_{\frac{3}{2}} N \rceil$ bound. An $N = 7$ AllReduce is not a good test case for the elimination method, because it cannot be applied with any decomposition and therefore is near equivalent in runtime as the recursive doubling in Fig. 20.

6.5. Validation

The purpose of the simulator was to understand the recursive multiplying algorithm more thoroughly than with just the theoretical and experimental aspects. The simulator in the current state is not validated and cannot replicate the experimental data discussed in Section 5. This is due to the simplicity of the model, it does not model any interaction between processes other than tasks. The pipelining postal model is a simple addition to the standard postal model which in itself is a simple theoretically based

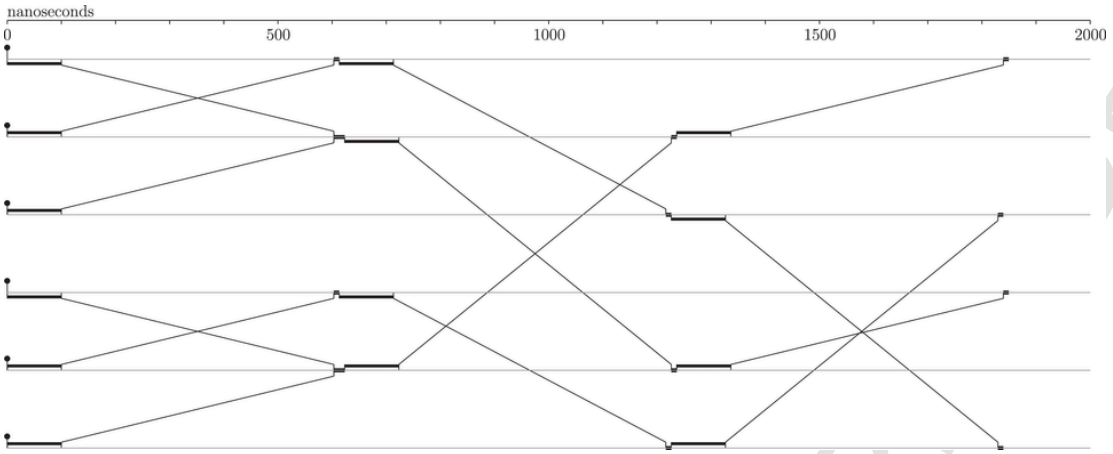


Fig. 23. Simulator timeline of a non-overlapping elimination protocol with six processes.

communications model. Experiments within the simulator have been done with respect to topology and simple uniform noise, but neither of these additional components yielded results replicating the experimental results.

7. Related work

Motivated by the LoP and LogfP model, Hoefler et al. [11] introduced a barrier operation based on the n-way dissemination pattern which allows for higher performance. The n-way dissemination pattern is an extension of the original dissemination pattern also used for a barrier operation [6,10]. By allowing a process to send multiple messages the scaling of the dissemination barrier is improved from $O(\log_2 N)$ with the n-way dissemination barrier.

End et al. [9] introduced the n-way dissemination AllReduce which allows for a large improvement on InfiniBand. When $N \neq (n+1)^k$ there is potential for duplication. An adaption is presented which performs a post process when duplication occurs, based on the data boundary from the previous stage. This allows for the correct result to be computed. Since butterfly-like patterns require an associative operator, this algorithm is only suitable for a subset of use cases.

8. Future work

The recursive multiplying algorithm is currently only implemented in the benchmark presented in Section 5. An implementation for the Edinburgh MPI for Research Library (EMPI4Re) is planned, which will include prime merging alongside the generalized fix. Since larger messages are expected in the context of MPI, an exploration will be performed to evaluate the capabilities of the Cray Aries [5] network such that an appropriate threshold can be used for the recursive multiplying method. Finally, the EMPI4Re implementation will be compared directly to MPICH.

Currently the schedule which is experimentally determined or analytically found is assumed to be order invariant except for the collapse and expand stages. While this is reasonable on a network such as the Cray Aries [5], which is very close to an all-to-all network, on different networks such as fat-trees the ordering could be important. By performing stages in a specific order dependent on the network, traffic can potentially be reduced. This is an obvious extension to allow for shared memory, on-node operations. On ARCHER [1] this would imply a 24-way all-to-all, since it can be performed efficiently, which reduces the costs associated with this operation. In addition, exploring the behaviour of message pipelining on the Infiniband network would be interesting with recursive multiplying. Finally, evaluating schedules using an efficient heuristic would be required for large-scale computers to determine the best schedule to use, since an exhaustive search would be prohibitively expensive.

Finally, the simulator has several features which have to be added to achieve a validated state. Better control of topology, location and noise are important aspects to be able to replicate, however congestion is likely the largest contribution to the given results. In addition, an exploration of self-congestion of the recursive multiplying algorithm is important since the algorithm purposefully floods the network with as many messages as possible.

9. Conclusions

We presented the recursive multiplying algorithm, which is a generalisation of the state of the art recursive doubling with pairwise exchange algorithm used in MPICH for small message AllReduce operations. We showed experimental results of reductions in execution times of 8% to 40% with recursive multiplying compared to recursive doubling on ARCHER, a Cray XC30.

Using message pipelining we replaced the original pairwise exchange with a multi-way exchange allowing small message size AllReduce operations to be performed faster. With this extension we were able to construct AllReduce operations with variable *b* values to accelerate large-scale operations with special cases only for large prime factors, instead of for non-power-of-two cases.

If future hardware designs support a higher degree of message pipelining, the time for AllReduce operations could be further improved due to the increase in the fanout of the algorithm. This would decrease the critical path of the algorithm or increase the range across the machine in the same number of stages. A corresponding increase in bandwidth capability would be required of the networks to support the increased number of messages.

Acknowledgments

We would like to thank Daniel Holmes for the help regarding the evaluation strategy and Justs Zarins for various coffee discussions.

This work was supported by grant EP/L01503X/1 for the University of Edinburgh School of Informatics Centre for Doctoral Training in Pervasive Parallelism (pervasiveparallelism.inf.ed.ac.uk) from the UK Engineering and Physical Sciences Research Council (EPSRC) and by the European Commission through the EPiGRAM project (Grant Agreement no.610598, www.epigram-project.eu). The work has made use of resources provided by the Edinburgh Parallel Computing Centre (EPCC, http://www.epcc.ed.ac.uk).

References

[1] ARCHER. http://www.archer.ac.uk/. Accessed: 2016-04-18.
[2] MPI: A Message-Passing Interface Standard. https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf. Accessed: 2016-05-09.
[3] MPICH. http://www.mpich.org/. Accessed: 2016-04-18.
[4] Using the GNI and DMAPP APIs. http://docs.cray.com/books/S-2446-5202/S-2446-5202.pdf. Accessed: 2016-04-18.
[5] B. Alverson, E. Froese, L. Kaplan, D. Roweth. Cray ® XC Series Network. 1–28.
[6] E.D. Brooks, The butterfly barrier, Int. J. Parallel Program. 15 (4) (1986) 295–307.
[7] E. Chan, M. Heimlich, A. Purkayastha, R. van de Geijn, Collective communication: theory, practice, and experience, Concurrency Comput. 19 (13) (2007) 1749–1783.
[8] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, T. von Eicken, Logp: towards a realistic model of parallel computation, Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 28, 19931–12.
[9] V. End, R. Yahyapour, C. Simmendinger, T. Alrutz, Adaption of the n-way dissemination algorithm for GASPI split-phase allreduce, The Fifth International Conference on Advanced Communications and Computation, 201513–19.
[10] D. Hensgen, R. Finkel, U. Manber, Two algorithms for barrier synchronization, Int. J. Parallel Program. 17 (1) (1988) 1–17.
[11] T. Hoefler, T. Mehlan, F. Mietke, W. Rehm, Fast barrier synchronization for infiniband, 20th International Parallel and Distributed Processing Symposium, IPDPS, 2006.
[12] T. Hoefler, T. Mehlan, F. Mietke, W. Rehm, Logfp - a model for small messages in infiniband, Parallel and Distributed Processing Symposium, 2006.
[13] T. Hoefler, W. Rehm, A communication model for small messages with infiniband, PARS Mitteilungen, 200532–41.
[14] T. Hoefler, T. Schneider, A. Lumsdaine, LogGOPSim: simulating large-scale applications in the logGOPS model, Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, 2010597–604.
[15] H. Pritchard, I. Gorodetsky, D. Buntinas, A uGNI-based MPICH2 nemesis network module for the cray XE, Recent Adv. Message (2011) 110–119.
[16] R. Rabenseifner, Automatic mpi counter profiling, 2000. 42nd CUG Conference.
[17] R. Rabenseifner, J.L. Träff, More efficient reduction algorithms for non-power-of-two number of processors in message-passing parallel systems, Recent Advances in Parallel Virtual Machine and Message Passing Interface 3241 (2004) 36–46.
[18] R. Thakur, W. Gropp, Improving the performance of collective operations in MPICH, Recent Adv. Parallel Virtual Mach. Message Passing Interface 2840 (2003) 257–267.
[19] R. Thakur, R. Rabenseifner, W. Gropp, Optimization of collective communication operations in MPICH, Int. J. High Perform. Comput. Appl. 19 (1) (2005) 49–66.